Approximate Shortest Path on a Polyhedral Surface Based on Selective Refinement of the Discrete Graph and Its Applications

Takashi Kanai
RIKEN
Materials Fabrication Lab.
.

Hiromasa Suzuki
University of Tokyo
Dept. Precision Engineering
.

Abstract

A new algorithm is proposed for calculating the approximate shortest path on a polyhedral surface. The method mainly uses Dijkstra's algorithm and is based on selective refinement of the discrete graph of a polyhedron. Although the algorithm is an approximation, it has the significant advantages of being fast, easy to implement, high approximation accuracy, and numerically robust. The approximation accuracy and computation time are compared between this approximation algorithm and the extended Chen & Han (ECH) algorithm that can calculate the exact shortest path for non-convex polyhedra. The approximation algorithm can calculate shortest paths within 0.4% accuracy to roughly 100-1000 times faster than the ECH algorithm in our examples. Two applications are discussed of the approximation algorithm to geometric modeling.

Keywords*: geometric modeling, computational geometry, polyhedral surface, shortest path, Dijkstra's algorithm*

1. Introduction

Polyhedral surfaces, especially meshes that consist of planar triangle faces, are the fundamental geometric representation in computer graphics and related areas. Those surfaces with a large number of faces (called *dense*) can now be generated more easily due to the recent development of range scanners, GPS, and so on. For example, surface reconstruction from range images [3] and the use of such dense polyhedral surfaces have been active research topics for computer graphics applications. Geographic information systems (GIS) usually use triangular irregular networks (TINs) which are polyhedral representations of geographical features.

Finding the shortest path on such polyhedral surfaces is a fundamental problem in computational geometry, and becomes an important technique for the application of robotics, GIS, route finding, and so on. The purpose of this paper is to establish an efficient algorithm for calculating shortest paths and to use them for geometric modeling. There are many algorithms for finding the shortest path on 2D polygons and 3D surfaces, and in 3D spaces. However, the algorithms for finding the *exact* shortest path on a polyhedral surface (including the non-convex case) [17, 2] usually involve high time and space costs. It is therefore not practical to apply these algorithms to a dense polyhedral surface in most cases.

Instead, we have focused on finding the *approximate* shortest path [11, 15]. The algorithm proposed in this paper for a polyhedral surface (possibly including the non-convex case) mainly uses Dijkstra's algorithm and is based on selective refinement of the discrete graph of a polyhedron. That is, Dijkstra's algorithm is iteratively used to narrow the region in which the shortest path can exist. Our algorithm has some significant advantages: it is fast and easy to implement, it has high accuracy, and needs less space cost to get an actual path. To show some features of our algorithm, we compare the approximation accuracy and computation time between our algorithm and the Chen and Han (CH) algorithm. The CH algorithm is the fastest one for finding the exact shortest path. Unfortunately, it sometimes fails to compute for a non-convex polyhedron. In this work, we extend the CH algorithm so that it can be used for more general cases.

We also discuss applications of our algorithm to geometric modeling. It is to be noted that our algorithm can only find the approximate shortest path, so it is not suitable for applications which absolutely need the exact path (or only its length). We show two examples of our algorithm being efficiently used for modeling: One involves specification of the boundaries of a region on a polyhedral surface. It is not necessary for such boundaries to be exact paths, but they do need to be smooth and to be calculated rapidly and efficiently. The other involves the interactive length measurement of a geodesic curve on a 3D model in a case that also

does not need to be exact.

2. Shortest Path Problem on a Polyhedral Surface

A survey of the shortest path problem concerning a two or higher dimensional geometric object (a surface, space, network, etc.) can be found in [16]. We mainly discuss here about finding the shortest path between two points on a polyhedral surface.

An important property of the shortest path on a polyhedron is its local optimality called *unfolding*: the path must enter and leave at the same angle to the intersecting edge. It follows that any locally optimal sub-path joining two consecutive obstacle vertices can be unfolded at each edge along its edge sequence, thus obtaining a straight segment. This property was used by Sharir and Schorr [19] who first proposed an $O(n^3 \log n)$-time algorithm (n is the number of edges) to find the exact shortest path on a convex surface. Mitchell et al. [17] have obtained an $O(n^2 \log n)$-time algorithm for general polyhedra by developing a continuous Dijkstra method for propagating the shortest path map over a surface. While Chen and Han [2] subsequently improved this to an $O(n^2)$-time algorithm, faster algorithms than these cannot presently be found.

On the other hand, some other work has been done on the approximate shortest path. This has involved less computation times than the exact approaches, but it is a trade-off with the path accuracy. In general, the term $(1+\epsilon)$-*approx.* is usually used for evaluating the computation time for these approximate approaches, meaning that a calculated approximate path guarantees the accuracy within (1+ϵ) times longer than the exact path. In this evaluation, smaller ϵ results in better accuracy.

Algorithms theoretically exist that are fast and require less space if limited to convex polyhedra. For example, the approach proposed by Har-Peled [6] is based on the construction of a tight bounding volume covering a polyhedron. By performing $O(n)$ pre-processing, the $O((\log n)/\epsilon^{3/2} + 1/\epsilon^3)$-time computation of a $(1+\epsilon)$-approx. shortest path is possible. In the case of general polyhedra, Varadarajan and Agarwal [21] have proposed algorithms that compute a 13-approx. (ϵ =12) path in $O(n^{5/3} \log^{5/3} n)$-time or a 15-approx. (ϵ =14) path in $O(n^{8/5} \log^{8/5} n)$-time. These are based on partitioning the surface into $O(n/r)$ patches, each having at most r faces. However, no approximations with high accuracy can be obtained by these approaches.

Some practical methods for finding the approximate shortest path based on subdivided discrete graph searching have recently been proposed. The algorithm described in this paper is of this type. The common characteristic of these approaches (including ours) is to create a certain graph (called an *edge subdivision graph* or a *pathnet*) which is needed for path computation as a pre-processing step.

The $O(n \log n)$-time approach by Lanthier et al. [11] first adds some intermediate points to edges, and discrete graph G is created by using these points and the original vertices. Then Dijkstra's algorithm is applied to G. After picking up the faces under the calculated shortest path of G (called the *sleeve*), the $O(\log k)$ algorithm (k is the number of edges passed by the shortest path) proposed by Guibas and Hershberger [5] is used to refine the graph. However, [5] uses a rather special data structure called the *hourglass*, and its algorithm is difficult to implement. Mata and Mitchell [15] have proposed another approach called the pathnet algorithm which is applied with, at worst, $O(kn^3)$-time (where k is the number of rays for each vertex). One practical aspect of this algorithm is the robustness of the computation. This algorithm is sensitive to numerical errors because it requires solving a fourth-degree polynomial. An another aspect is the path accuracy, a large number of k being required to obtain an approximate shortest path close to the exact solution.

The algorithm described in this paper is similar to the algorithm of Lanthier et al. [11]. The major difference is that our algorithm is iterative: the shortest path is calculated by selective refinement of the subdivided discrete graph of a polyhedral surface.

3. Approximate Shortest Path Computation

A *polyhedral surface* is a connected union of a finite number of polygonal faces, with any two polygons intersecting along a common edge, at a common vertex, or not at all, and each edge belonging exactly to a polygon. Given polyhedral surface $\mathcal{M}$, shortest path $\mathcal{L}$ from source vertex V_S to destination vertex V_D on $\mathcal{M}$ is defined by:

$$\begin{aligned} &\mathcal{L} = \{v_1, v_2, \ldots, v_n, e_1, e_2, \ldots, e_{n-1}\}, \\ &V_S = v_1, \quad V_D = v_n, \quad e_i = \{v_i, v_{i+1}\}, \end{aligned} \tag{1}$$

where v_i and e_i denote a point and an edge on a face of $\mathcal{M}$, respectively.

Let $l = |\mathcal{L}|$ be the *length* of shortest path $\mathcal{L}$. Then:

$$l \;=\; |\mathcal{L}| = |e_1| + |e_2| + \ldots + |e_{n-1}|, \tag{2}$$

where $|e_i|$ is the Euclid distance between two connected vertices v_i and v_{i+1} of edge e_i.

3.1. Pre-processing for path calculation

Pre-processing involves creating initial discrete graph $G^0(v, e)$ from $\mathcal{M}$. $G^0(v, e)$ is composed of vertices / edges of $\mathcal{M}$ and additional vertices / edges.

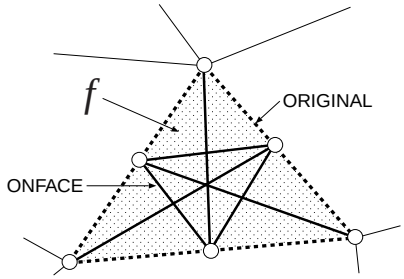


Figure 1. Edge classifications

We define here some classifications to edges on $\mathcal{M}$ to judge whether an edge should be added or not during the process of the algorithm: ORIGINAL refers to the original edges or their subdivided edges of $\mathcal{M}$, and ONFACE refers to the others, as shown in Figure 1.

G^0 is next created. First, the vertices and edges of $\mathcal{M}$ are added to G^0. Then additional vertices are sampled on each edge of $\mathcal{M}$. These are called *Steiner Points* (SPs). The number of added SPs on each edge are decided by its length and parameter γ. Let $|e|$ be the length of edge e. Then the number of SPs on e is decided as $(|e| \bmod \gamma) - 1$ so that smaller γ creates more SPs. Note that γ is a size-dependent value. To be independent of the size of $\mathcal{M}$, $\mathcal{M}$ is scaled into a unit cube in advance so that the edge length is normalized. γ is a user-defined value that decides the trade-off between the approximation accuracy and the computing time and space.

Intermediate edges are created in G^0 by using the original and additional vertices. If pair of vertices $\langle v_a, v_b \rangle \in G^0$ satisfies either 1. or 2. in **Condition 1** bellow, edge $e = \{v_a, v_b\}$ is added to G^0:

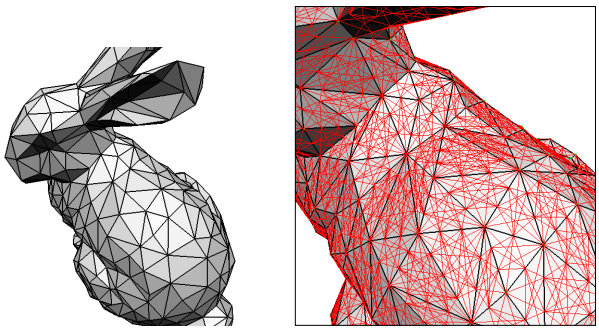
Condition 1

1. $\langle v_a, v_b \rangle$ is a pair of vertices in G that are on different edges of a face.
2. $\langle v_a, v_b \rangle$ is a pair of vertices in G that are next to each other on the same edge.

Figure 2(b) illustrates a part of initial discrete graph G^0 (2,859 vertices, 16,055 edges) of the "bunny" model (525 vertices, 999 edges) shown in Figure 2(a). In Figure 2(b), thin lines denote additional edges. We have set γ to 0.04 in this example, the average number of SPs per edge is 1.52.

3.2. Approximate Shortest Path Algorithm

By using G^0, shortest path $\mathcal{L}$ between V_S and V_D is calculated by the following algorithm. Our algorithm repeats **STEP 1** - **STEP 4**, incrementing loop counter i from 0 by 1 each time.



(a) (b)

Figure 2. Initial discrete graph G^0

STEP 1: Calculate shortest path on G^i

Shortest path $\mathcal{L}^i$ for discrete graph G^i and its length l^i are calculated by using Dijkstra's algorithm. The shortest path is composed of vertices and edges in G^i.

STEP 2: Generate new graph G^{i+1} from path $\mathcal{L}^i$

Shortest path $\mathcal{L}^i$ is traced and its component vertices are added to G^{i+1}. Other connected vertices of edges including these path vertices are also added if the corresponding edges are ORIGINAL. For pair of vertices $\langle v_a, v_b \rangle \in G^{i+1}$, edge $e = \{v_a, v_b\} \in G^i$ is added to G^{i+1}.

STEP 3: Add SPs and edges to G^{i+1}

m SPs are defined for edges $e \in G^{i+1}$ at even intervals in G^{i+1} if e is ORIGINAL. If pair of vertices $\langle v_a, v_b \rangle \in G^{i+1}$ satisfies both $e = \{v_a, v_b\} \notin G^{i+1}$ and either 1. or 2. in **Condition 1**, e is added to G^{i+1}. e is ONFACE if $\langle v_a, v_b \rangle$ satisfies to 1. in **Condition 1**, else e is ORIGINAL. When e is ORIGINAL, its original edges are deleted from G^{i+1}.

STEP 4: Update the graph

$G^i \leftarrow G^{i+1}$

When **STEP 1** is finished, the length of shortest path l^i of graph G^i and of l^{i-1} of its previous graph G^{i-1} are compared. The algorithm is completed if $|l^i - l^{i-1}| < \epsilon$. The addition of SPs and edges to G^0 described in the previous subsection corresponds to **STEP 3**, but the number of SPs

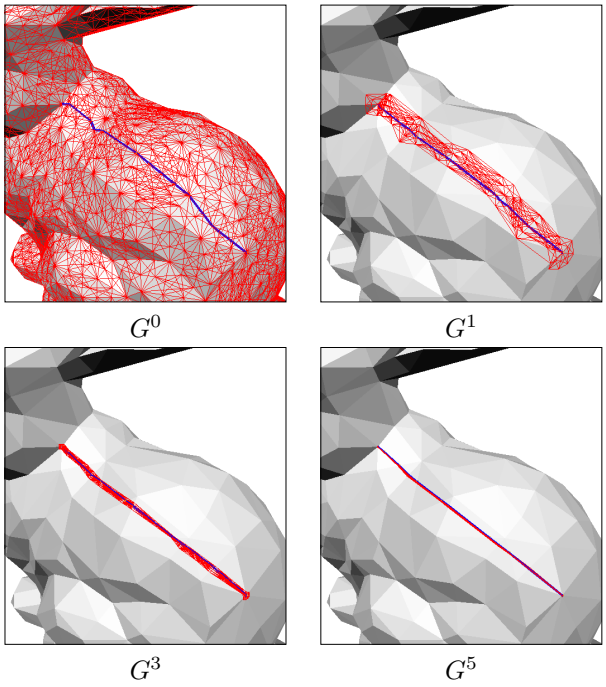


Figure 3. Iterations of the algorithm

per edge m is different. m is another variable that should be decided by the user. It is a trade-off between the decrease in the number of iterations and the increase in computing time and space. While it is also possible to set m to a different value per edge, we set m as a constant value for all edges because the length of edges in G^0 is almost uniform throughout the pre-processing stage.

The characteristic of this algorithm are presented in Figure 3 which shows the state for calculating the shortest path between two given vertices of the bunny model in Figure 2. We set $\gamma = 0.04, m = 1$ for this example, and the algorithm terminated after seven iterations. In Figure 3, the sequences of discrete graphs G^0, G^1, G^3, G^5 and the shortest path of each graph calculated in **STEP 1** are shown. The thin lines in Figure 3 represent edges of a graph calculated in **STEP 2** and **STEP 3**, while bold lines show the shortest path calculated on that graph. It can be seen that the region in which a path exists gradually becomes narrower and finally converges to a poly line.

Table 1 shows the number of elements (vertices, edges) on G^i, the length of shortest path l^i and the sum of each processing time for **STEP 1 - STEP 4**. Our algorithm searches almost a constant number of edges (from 350 to 400 in this example) on each graph G^i, except for G^0 in the first stage which has a large number of edges. It can be also seen that the total time needed for processing our algorithm largely depends on that for calculating the shortest path on G^0 in

	$\|v\|$	$\|e\|$	l^i	T (sec.)
G^0	2,859	16,055	0.52200	0.083
G^1	79	361	0.51408	0.017
G^2	75	365	0.51277	0.017
G^3	76	371	0.51252	0.017
G^4	78	393	0.51237	0.017
G^5	78	393	0.51234	0.017
G^6	70	313	0.51234	0.017

Table 1. Number of elements, length of the shortest path and processing time for each graph by our algorithm

STEP 1 by using Dijkstra's algorithm. We used a simple method for this problem based on a priority queue of partial-order trees that can be processed with $O(n \log n)$-time [1]. As this method evaluates the length of each edge, its processing time can be decided by the number of edges in G^0 and thus by γ.

To implement our algorithm, we prepared a simple graph structure for representing G, consisting of nodes for vertices and SPs, and links for (subdivided) edges of $\mathcal{M}$ and additional edges. Each node has a 3D coordinate value and a pointer to its connected links. A link has pointers to two end nodes, has its length, and has a flag for its type (ORIGINAL or ONFACE). If a node or a link comes from the original element of $\mathcal{M}$, it has a pointer to that element. If a link is ONFACE, the link has a pointer to a face of $\mathcal{M}$ under it.

4. Experimental Results and Discussion

We evaluate in this section our approximate shortest path algorithm from two viewpoints: one is the approximation accuracy and computation time with different γ and m; the other is the numerical accuracy. We also made a comparison between our algorithm and the exact computation of the shortest path on convex polyhedra as proposed by Chen and Han [2]. We discuss this algorithm in the first subsection.

4.1. Extended CH Algorithm

To evaluate to our approximate path algorithm, we implemented the Chen and Han algorithm [2] (we call it the *CH algorithm* from now on) with an extension so that it can compute the exact shortest path between any two vertices on a non-convex polyhedral surface.

The CH algorithm improves on the continuous Dijkstra method proposed by Mitchell et al. It utilizes the unfolding property (described in Section 2), whereby the shortest path and its length can be calculated by constructing a sequence

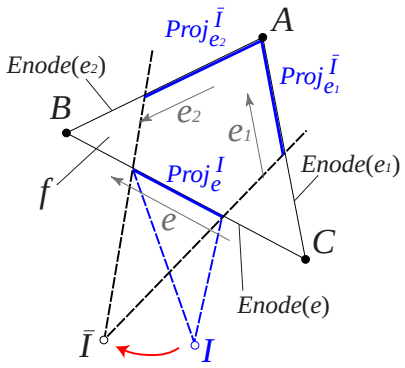


Figure 4. Relationship between $Enode(e)$ and its children in the CH algorithm

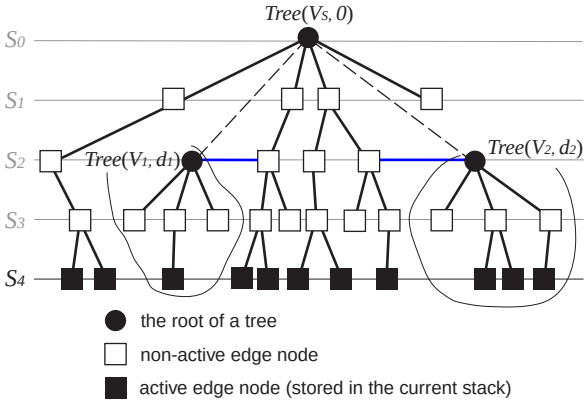


Figure 5. Construction of sequence trees in the ECH algorithm

tree structure. Its root is start vertex V_S, and leaf nodes are oriented edges. Edge node $Enode(e) = (e, I, Proj_e^I)$ is composed of oriented edge e, image I of V_S, and its projection $Proj_e^I$ to an edge. $Enode(e)$ has at most two child edge nodes. Figure 4 illustrates the relationship between $Enode(e)$ and its children $Enode(e_1), Enode(e_2)$. Each child of $Enode(e)$ can only be defined if projection $Proj_{e_1}^{\bar{I}}, Proj_{e_2}^{\bar{I}}$ of $\bar{I}$ to e_1, e_2 exists. $\bar{I}$ is calculated by rotating I around e so that $\bar{I}$ and face f are co-planar. It is said that $Enode(e)$ *covers* vertex A if both $Proj_{e_1}^{\bar{I}}$ and $Proj_{e_2}^{\bar{I}}$ exist. In addition, $Enode(e)$ *occupies* A if $|\bar{I}A|$ (the Euclid distance between A and $\bar{I}$) is the shortest. In this case, A has vertex node $Vnode(A) = (Enode(e), \bar{I}, A)$. The shortest path from V_S to V_D is calculated by tracing the edge node's parents from $Vnode(V_D)$.

The CH algorithm can always be successfully applied if a polyhedral surface is convex. For the general case, including a non-convex surface, there are some cases of V_D not having $Vnode(V_D)$ when the algorithm is terminated. The shortest path cannot be calculated in these cases. [2] points out that the shortest path of a non-convex polyhedron may pass some vertices [17] except V_S and V_D, and that it is a good choice to grow a sub-tree from a passed vertex. Unfortunately, no details for growing a sub-tree are mentioned in [2]. The problem is *how* or *when* we can begin to grow a sub-tree, because we cannot judge the vertex that the shortest path passes through during the process of the CH algorithm.

An extension for growing a sub-tree is presented next. We call it the *extended CH (ECH) algorithm*. Figure 5 illustrates sequence trees of the ECH algorithm. Root tree $Tree(V_S, 0)$ at V_S has a sequence of sub-trees. In Figure 5, such sub-trees correspond to $Tree(V_1, d_1)$ and $Tree(V_2, d_2)$. $Tree(V, d)$ has many pointers: one to the edge nodes (their number is that of the faces adjacent to V), one to the vertex nodes (their number is that of the edges adjacent to V), one to vertex V, geodesic distance d between V to V_S (this can be calculated during the process of the algorithm) and one to a parent tree. Each of the edge nodes belongs to the same tree as its parent node. A sub-tree is created if the edge node occupies a vertex V. If V with a sub-tree is occupied by another edge node, a sub-tree (including child sub-trees) and its child edge nodes are all deleted, and a new sub-tree is created.

Figure 6 shows the pseudo-code of the ECH algorithm. There are two major differences from the original: (1) the main loop is controlled by using stack S that stores edge nodes; (2) when it is judged that an edge node occupies A, A's sub-tree (including child sub-trees) are all deleted, and a new sub-tree is created. The time complexity keeps $O(n^2)$, but its efficiency is worse. The space complexity also keeps $O(n)$ if only the length of the path is needed, or $O(n^2)$ if the path itself is needed.

4.2. Approximation Accuracy and Computation Time

We next investigate how different values for user-specified variables γ and m influence the approximation accuracy and computation time. We used the simplified model of a "bunny" shown in Figure 7. To simplify the original model, we used the quadric error metric (QEM) based approach proposed by Garland and Heckbert [4]. It was difficult to use the original model because our naive implementation of the ECH algorithm needed quite a large amount of space to store the shortest path information. Figure 7(b) shows a simplified model in which the number of faces is 9,999.

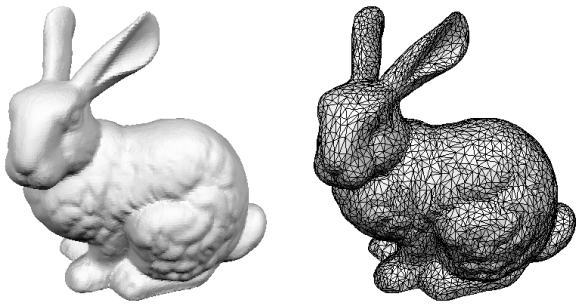
```
Procedure Extended_CH_Algorithm
begin
  Create_Tree(V_S, 0, S);
  while S ≠ nil do
    Enode(e) ← top of S;
    unfold I to Ī; calculate Proj^Ī_{e1} and Proj^Ī_{e2};
    if Enode(e)'s shadow covers A then
      Enode' ← previously occupied enode;
      if Enode(e) occupies A over Enode' then
        clip off Enode's two children that is
          impossible to define the shortest sequence
          and their child enodes;
        if A has a tree then delete A's tree; end if
        insert Enode(e1) and Enode(e2) to
          Enode(e) as its children and to S';
        Create_Tree(A, d_A, S');
      else  // Enode(e) doesn't occupy A
        insert either Enode(e1) or Enode(e2) that is
          possible to define the shorest sequence;
      end if
    else  // Enode(e)'s shadow doesn't cover A
      insert either Enode(e1) or Enode(e2) that is
        not empty to Enode(e) as its child and to S';
    end if
    S ← S';
  end while
end

Procedure Create_Tree(V, d_V, S)
begin
  create Tree(V, d_V);
  forall edges e opposite to V do
    insert Enode(e) to Tree(V, d_V) as its child
      and to S;
  end for
end
```

Figure 6. Pseudo-code for the ECH algorithm

We obtained the graph data needed for the evaluation. First, 1000 pairs of vertices from the models in Figure 7(b) were randomly selected. The shortest paths and their lengths for these pairs are calculated by the ECH algorithm and by our approximation algorithm with different γ and m. All our implemention was conducted on a graphics workstation (CPU: MIPS R10000, 175 MHz).

Figure 8(a) shows a comparison of the approximation accuracy for the model in Figure 7(b). The vertical axis denotes the percentage of the average length of the shortest path of 1000 pairs to that by the ECH algorithm. Figure 8(b) shows a comparison of the computation time for the model in Figure 7(b), the vertical axis denotes the average computation time for the shortest paths. The horizontal axis in each of Figures 7(a)-(b) denotes the value corresponding to γ. Instead of using γ, which is hard to intuitively understand, we used the number of SPs per edge which was calculated from γ.



(a) (b)

Figure 7. The original "bunny" model (35,947 vertices, 69,451 faces) and its simplified model (5,047 vertices, 9,999 faces)

It can be seen from Figure 8(a) that the average lengths of the approximate shortest paths for all γ and m are within a 0.4% approximation accuracy from that of the exact shortest path. The length of a path with smaller γ approaches that of the exact path, m giving some contribution to improving the approximation accuracy. It can also be seen from Figure 8(b) that the computation time with our approximation algorithm is much less than that by the ECH algorithm (170.245 Sec.). The computation time increases with smaller γ or larger m. In particular, γ has a larger effect on the computation time than m. The computation time of a path with larger γ increases in the order of $O(n \log n)$.

Figure 8(c) shows a graph that represents the relationship between the length of the shortest path calculated by our approximation algorithm and the computation time. The horizontal axis denotes the path length, and the vertical axis denotes the computation time. To draw this graph, we classified the results into five groups according to the path length. The length interval for each group is 0.2, and we plotted the weighted average length of each group. The graph represents the characteristics of our algorithm very well. The computation time depends on the length of a path. The computation time for paths with different lengths varies in the order of $O(k \log k)$ (k is the number of edges passed by the shortest path) because our algorithm repeats the Dijkstra's algorithm for a discrete graph generated from a path. Figure 8(d) shows a graph of the relationship between the number of iterations and m. We can see from this figure that larger m decreases the number of iterations.

It can be seen from all the information in Figure 8 that our approximation algorithm can calculate the shortest path within a 0.4% accuracy roughly 100-1000 times faster than the ECH algorithm. While both γ and m contribute to the approximation accuracy to a certain extent, it is not always

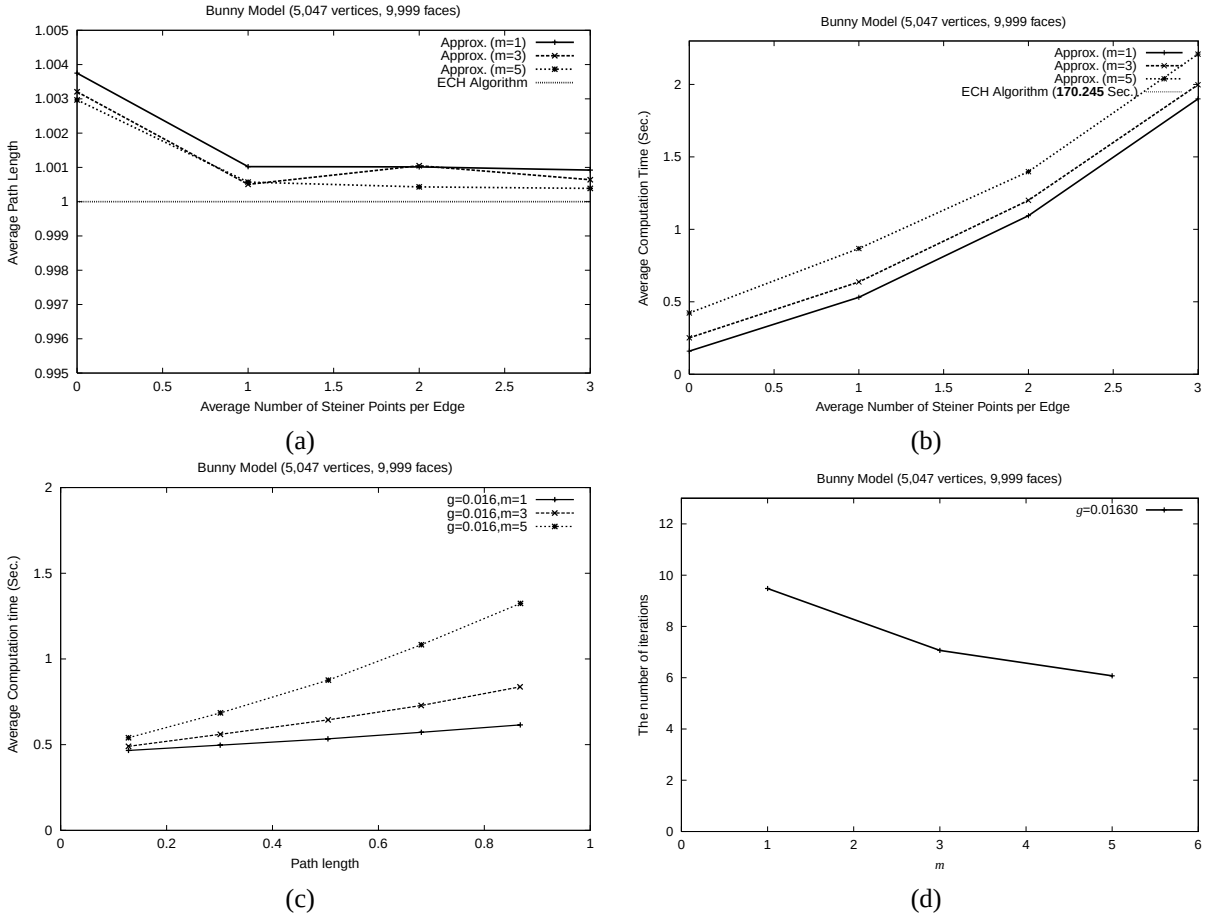


Figure 8. (a)-(b) Relationship between the path accuracy and the computation time (c) Relationship between the path distance and the computation time (d) Relationship between the number of iterations and m

the case that smaller γ or larger m gives rise to the efficiency of our algorithm because the computation time is also increased.

Furthermore, we conducted the same experiments for more simplified models in which the number of faces were about 1,000 and 5,000 respectively. Almost same results as those already described were obtained, so they are not further reported in this paper.

4.3. Numerical Accuracy

We now present experimental data about measurement of the perimeter length of regular n-gonal prisms. This data represents the superiority of our approximation algorithm over the ECH algorithm in terms of its numerical accuracy. Figure 9(a) shows a development of a regular n-gonal prism, and Figures 9(b)-(d) show regular n-gonal prisms in which $n = 10, 100$ and 1000, respectively. Perimeter length d of a regular n-gonal prism is $d = 2n \sin \frac{\pi}{n}$. In this experiment, we compare an analytical solution for d to solutions by the ECH algorithm and by our approximation algorithm.

Figure 10 shows the result of d calculated by the analytical method and by these algorithms for some sampled values of n from 0 to 15,000. You can see that the solutions by our algorithm are almost the same as those by the analytical method, while the solution by the ECH algorithm deviates greatly from it when n exceeds 3,000. This deviation originates from the method for calculating a path and its length by the ECH algorithm, which uses the 3D rotation computation and accumulates numerical errors. Our approximation algorithm merely uses the addition of the length of edges in discrete graphs. Consequently, our approximation algorithm is demonstrably stable in terms of its numerical accuracy.

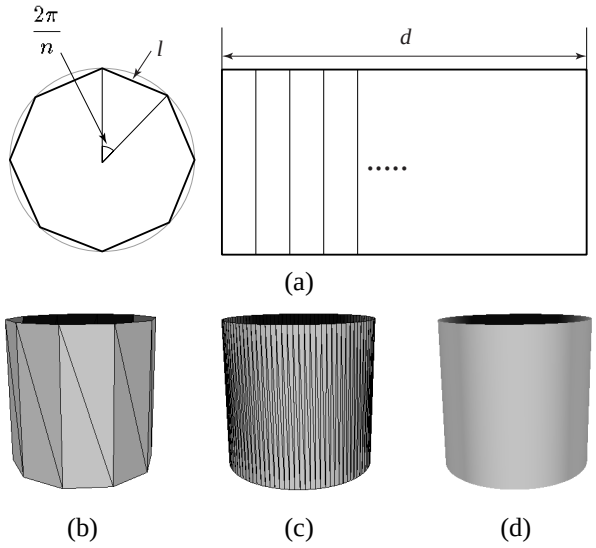


Figure 9. Regular n-gonal prisms for evaluating the numerical accuracy

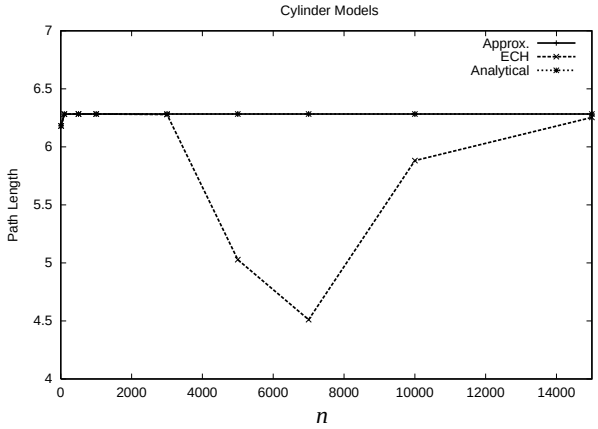


Figure 10. Comparison of numerical errors by analytical solution, our approximation algorithm and the ECH algorithm

5. Applications

We applied our approximation algorithm to two applications, one being as a tool for specifying the local region on a polyhedral surface, and the other for interactively measuring the geodesic distance on a polyhedral surface.

5.1. Region Specification on a Polyhedral Surface

Operations which specify a region of dense polyhedral surfaces, especially those generated from range images, are needed for modeling, rendering and animation. Such examples are the addition of attributes such as texture [14], conversion to a parametric surface [9], local deformation of a polyhedral surface [12, 10, 13] and 3D morphing [7, 8]. We discuss here about the use of our approximation algorithm as a tool for interactively specifying the boundaries of local regions. It is desirable for the boundaries of these regions to be smooth poly lines. If a boundary is distorted, it will negatively effect the quality of the various results in such applications. Moreover, many trial-and-error operations are often forced to users. Therefore, the algorithms used in these operations should be sufficiently fast and efficient for such interactivity.

The application of our approximate shortest path algorithm to boundary specification first involves initial discrete graph G^0 being created. Efficient calculations are possible by preserving G^0 all the time during the input and modification of a boundary. Although our approximation algorithm needs only two vertices to define a path, it is desirable that the path is composed of more than just vertices for fine and accurate specification of the boundaries. A path is generated by defining some control vertices and then by calculating shortest paths between neighboring control vertices. Figure 11 shows an example of specifying a "star" region with many boundary lines on the belly of a "horse" model (19,851 vertices, 39,698 faces). Spheres in this figure denote control vertices specified by the user. The input or modification of each control vertex leads to re-calculation of the two paths neighboring it, but this re-calculation is fast enough to make it possible to specify interactively.

5.2. Measurement of Geodesic Distance

Finding the shortest path to measure geodesic distance on a polyhedral surface is one of the most important techniques for the application of robotics, GIS, route finding, and so on. These applications often require the path or its length to be accurate. A slight deviation from the exact path might have a bad influence on the application result. On the other hand, some applications certainly exist in which approximate solutions are fully acceptable. As an example of a medical application, measurement of the magnitude of a tumor of the brain by using 3D models reconstructed from a sequence of medical images such as those by CT or MRI can be considered. Our approximation algorithm is thus also suitable for applications that need interactivity and not an exact solution.

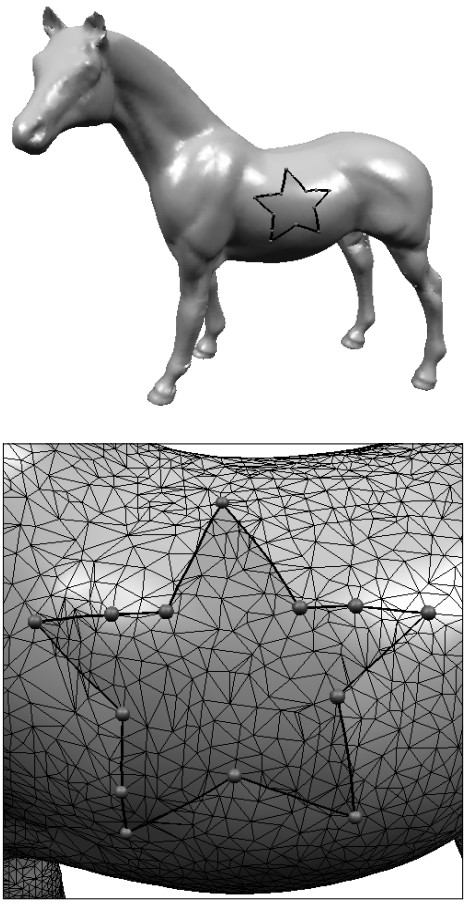


Figure 11. Generation of boundaries for specifying a "star" region

6. Conclusion and Future Work

This paper has proposed a new approach for calculating the approximate shortest path on a polyhedral surface based on selective refinement of the discrete graph. We have shown a comparison between our algorithm and the ECH algorithm in respect of the approximation accuracy, computation time and numerical accuracy. Two applications that suit the use of our approximation algorithm are discussed. The advantages of our approach are as follows:

- It is easy to implement. Only the routine of Dijkstra's algorithm and an additional graph structure are needed.
- It is fast. The computation time largely depends on a Dijkstra's algorithm. Our implementation executes in $O(n \log n)$-time. Thorup has recently proposed an $O(n)$-time method for processing Dijkstra's algorithm [20]. We need more consideration about this method.
- It provides high approximation accuracy. In our examples, an approximation accuracy within 0.4% was established.
- It is numerically robust. Our approximation algorithm merely uses the addition of the length of edges in discrete graphs.
- Although we have not described in this paper, it is also possible to calculate a weighted shortest path.

On the other hand, our algorithm has some disadvantages which we are working on to improve for the future work.

- Our approximation algorithm results in the shortest path of a *regular* polyhedral surfaces (for example, polygons arranged as a grid) being of low approximation accuracy. Such an example requires smaller γ to be set.
- The computation time depends on the length of a path, consequently, our algorithm cannot give a response with a constant time for a polyhedral surface. This may be an obstacle to some applications.
- The computation time in our algorithm largely depends on the time that searches a shortest path on a discrete graph G^0. To reduce the computation time, the use of more efficient search method alternative to Dijkstra's algorithm should be taken into consideration. For example, A* algorithm [18], one of the best-first search using the heuristic function, might be applicable.

Acknowledgement

Part of this research was supported by Elysium Co. Ltd. The bunny model was from Stanford University Computer Graphics Laboratory, and the horse model was from Cyberware Inc.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data structures and algorithms*. Addison-Wesley, 1983.

[2] J. Chen and Y. Han. Shortest paths on a polyhedron. In *Proc. 6th ACM Symp. on Computational Geometry*, pages 360–369, 1990.

[3] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *Computer Graphics (Proc. SIGGRAPH 96)*, pages 303–312, 1996.

[4] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *Computer Graphics (Proc. SIGGRAPH 97)*, pages 209–216, 1997.

[5] L. J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. *J. Computer and System Sciences*, 39(2):126–152, 1989.

[6] S. Har-Peled. Approximate shortest paths and geodesic diameters on convex polytopes in three dimensions. In *Proc. 13th ACM Symp. on Computational Geometry*, pages 359–366, 1997.

[7] T. Kanai, H. Suzuki, and F. Kimura. Metamorphosis of arbitrary triangular meshes with user-specified correspondence. *IEEE Computer Graphics and Applications*, 2000, to appear.

[8] T. Kanai, H. Suzuki, J. Mitani, and F. Kimura. Interactive mesh fusion based on local 3D metamorphosis. In *Proc. Graphics Interface '99*, pages 148–156, 1999.

[9] V. Krishnamurthy and M. Levoy. Fitting smooth surfaces to dense polygon meshes. In *Computer Graphics (Proc. SIGGRAPH 96)*, pages 313–324, 1996.

[10] S. Kuriyama and T. Kaneko. Discrete parametrization for deforming arbitrary meshes. In *Proc. Graphics Interface '99*, pages 132–139, 1999.

[11] M. A. Lanthier, A. Maheshwari, and J.-R. Sack. Approximating weighted shortest paths on polyhedral surfaces. In *Proc. 13th ACM Symp. on Computational Geometry*, pages 274–283, 1997.

[12] A. W. F. Lee, W. Sweldens, P. Schröder, L. Cowsar, and D. Dobkin. MAPS: Multiresolution adaptive parameterization of surfaces. In *Computer Graphics (Proc. SIGGRAPH 98)*, pages 95–104, 1998.

[13] S. Lee. Interactive multiresolution editing of arbitrary meshes. *Computer Graphics Forum (Eurographics 99)*, 18(3):73–82, 1999.

[14] J. Maillot, H. Yahia, and A. Verroust. Interactive texture mapping. In *Computer Graphics (Proc. SIGGRAPH 93)*, pages 27–34, 1993.

[15] C. S. Mata and J. S. B. Mitchell. A new algorithm for computing shortest paths in weighted planar subdivisions. In *Proc. 13th ACM Symp. on Computational Geometry*, pages 265–273, 1997.

[16] J. S. B. Mitchell. Geometric shortest paths and network optimization. In J. R. Sack and J. Urrutia, editors, *The Handbook of Computational Geometry*. Elsevier Science, 1998.

[17] J. S. B. Mitchell, D. M. Mount, and C. H. Papadimitriou. The discrete geodesic problem. *SIAM J. Computing*, 16(4):647–668, 1987.

[18] E. Rich. *Artificial Intelligence*. McGraw-Hill, 1983.

[19] M. Sharir and A. Schorr. On shortest paths in polyhedral spaces. *SIAM J. Computing*, 15:193–215, 1986.

[20] M. Thorup. Undirected single source shortest paths in linear time. In *Proc. 38th Symp. on Foundations of Computer Sciences*, pages 12–21, 1997.

[21] K. R. Varadarajan and P. K. Agarwal. Approximating shortest paths on an nonconvex polyhedron. In *Proc. 38th Annual Symp. on Foundations of Computer Science*, pages 182–193, 1997.